Efficient Index Learning via Model Reuse and Fine-tuning

Guanli Liu
The University of Melbourne, Australia
guanli.liu@student.unimelb.edu.au

Jianzhong Qi
The University of Melbourne, Australia
jianzhong.qi@unimelb.edu.au

Lars Kulik
The University of Melbourne, Australia
lkulik@unimelb.edu.au

Kazuya Soga
The University of Melbourne, Australia
ksoga@student.unimelb.edu.au

Renata Borovica-Gajic
The University of Melbourne, Australia
renata.borovica@unimelb.edu.au

Benjamin I. P. Rubinstein
The University of Melbourne, Australia
benjamin.rubinstein@unimelb.edu.au

***Abstract*—Learned indices using machine learning techniques have demonstrated potential as alternatives to traditional indices such as B-trees in both query time and memory. However, a well fitted learned index requires significant space consumption to train models and tune parameters. Furthermore, fast training methods—ones that train in one pass—may not learn the data distribution well. To consider both the fitness to data distribution and building efficiency, in this paper, we apply *pre-trained models* and *fine-tuning* to accelerate the building of learned indices by 30.4% and improve lookup efficiency by up to 24.4% on real datasets and 22.5% on skewed datasets.**

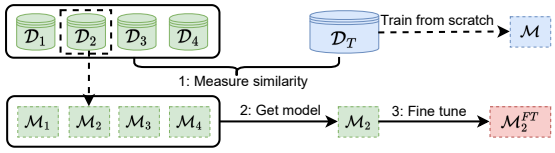
***Index Terms*—learned index, model reuse, fine-tuning c**

I. Introduction

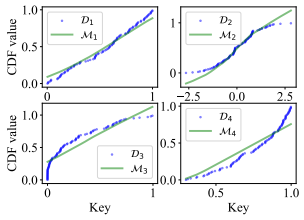
Learned indices – a technique that uses machine learning techniques as an alternative to database indices – has been shown to outperform traditional indices such as B-trees in both query time and memory consumption [1]–[3]. Given a dataset (e.g., a database table), an index is a structure that maps the index key $p.key$ of a data record p to its storage address $p.addr$. The idea of learned indices is to train a machine learning model $\mathcal{M}$ (e.g., a neural network or linear regression) to approximate the mapping from $p.key$ to $p.addr$, i.e., to learn the cumulative distribution function (CDF) of the dataset, assuming records are stored in key-sorted order [1]. The trained model $\mathcal{M}$ can predict $p.addr$ with a bounded error range $[err_l, err_u]$, i.e., the data record p can be found in the range of $[\mathcal{M}(p.key) - err_l, \mathcal{M}(p.key) + err_u]$ [3] with a *binary search*. Here, err_l and err_u are the minimal and maximal training errors of $\mathcal{M}$.

While learned indices have efficient query procedures, they can be prone to slow building, since machine learning models are expensive to train. The bottleneck of index building is that every data record within the training set will be fully scanned in each epoch with at least one epoch during learning.Even with simple models such as linear regression, a learned index such as the *recursive model index* (RMI) [1] is more than an order of magnitude slower to build than a B-tree [2]. Meanwhile, techniques that learn indices in a single pass such as *RadixSpline* [4] (RS) and PGM [3] still require a nested loop to build, e.g., to check the validity of given error bounds for every key. They also tend to produce sub-optimal indices of large sizes and lower query efficiency than RMI[1].

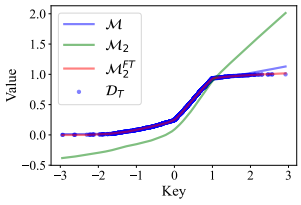
To address these issues, we aim to save index build times by leveraging pre-trained models to avoid training from scratch. This solution is inspired by *domain adaptation* [5]. Given a model $\mathcal{M}_S$ trained on a known (source) dataset $\mathcal{D}_S$, domain adaptation reuses $\mathcal{M}_S$ for a new (target) dataset $\mathcal{D}_T$ by fine-tuning $\mathcal{M}_S$ over $\mathcal{D}_T$ (see Fig. 1a). This avoids the expensive training of a new model on $\mathcal{D}_T$ from scratch.



(a) Pre-training: Model $\mathcal{M}_1$ to $\mathcal{M}_4$ are pre-trained on known datasets $\mathcal{D}_1$ to $\mathcal{D}_4$, respectively. Existing solutions train a new model $\mathcal{M}$ on $\mathcal{D}_T$ from scratch. We fine-tune a pre-trained model $\mathcal{M}_2$ as $\mathcal{M}_2^{FT}$ to index $\mathcal{D}_T$.



(b) The CDFs of four known datasets and their trained models.



(c) The CDF of $\mathcal{D}_T$ and models $\mathcal{M}$, $\mathcal{M}_2$, and $\mathcal{M}_2^{FT}$.

Fig. 1: Our pre-training and fine-tuning based approach.

For example, in Fig. 1a, there are four known (e.g., synthetic or historical) datasets $\mathcal{D}_1$ to $\mathcal{D}_4$ and we train a model over each one. We call the resultant models (i.e., $\mathcal{M}_1$ to $\mathcal{M}_4$) pre-trained models. If we train model $\mathcal{M}$ over $\mathcal{D}_T$ from scratch, the training cost is high. Instead, to save training time, we select $\mathcal{M}_2$ as the model to index $\mathcal{D}_T$ because $\mathcal{D}_2$ is the most similar to $\mathcal{D}_T$ among the four datasets. Subsequently, Fig. 1b shows the CDFs of the four known datasets and their corresponding models. Fig. 1c shows that $\mathcal{M}$ fits $\mathcal{D}_T$ better than $\mathcal{M}_2$ (which is the best fit among the four pre-trained models), while after

[1] https://learnedsystems.github.io/SOSDLeaderboard/leaderboard/

fine-tuning, the fine-tuned model $\mathcal{M}_2^{FT}$ almost perfectly fits $\mathcal{D}_T$. In addition, the cost of selecting and fine-tuning model $\mathcal{M}_2$ is substantially cheaper than training $\mathcal{M}$.

A key requirement for successful adaptation of $\mathcal{M}_S$ to $\mathcal{D}_T$ is that $\mathcal{D}_S$ and $\mathcal{D}_T$ should have similar distributions [6], [7]. Otherwise, $\mathcal{M}_S$ may yield large errors on $\mathcal{D}_T$. This motivates us to generate synthetic datasets to cover a wide range of different distributions and pre-train reusable models on such datasets. The next question is then how to select a pre-trained model for $\mathcal{D}_T$, i.e., how to measure the dataset similarity.

Our dataset generation process aims to simulate as many different CDFs as possible. We propose an efficient dataset generation method that takes a CDF distance threshold ϵ and a dataset cardinality n as the input, and it outputs a set of synthetic datasets. Then, we train a model $\mathcal{M}_S$ over every synthetic dataset $\mathcal{D}_S$. To measure the similarity between two datasets, we use a fast implementation of the *earth mover's distance* (EMD) [8]. When given a new dataset $\mathcal{D}_T$, to reuse a pre-trained model, we measure the EMD between $\mathcal{D}_T$ and all the synthetic datasets. We select a model $\mathcal{M}_S$ where $\mathcal{D}_S$ and $\mathcal{D}_T$ have the highest similarity (smallest EMD). Then, we fine-tune $\mathcal{M}_S$ over $\mathcal{D}_T$ to better fit the data distribution.

To showcase the applicability of our model reuse technique, we integrate it into RMI [1]—a learned index that displays state-of-the-art performance in key lookup speeds [9]. We show that model reuse and fine-tuning can significantly reduce the training time of the sub-models in RMI.

In summary, our key contributions are:

(1) We propose a model reuse and fine-tuning technique to accelerate index building and key lookup. The reused models are trained over synthetic datasets, which are generated based on a heuristic method.

(2) To reuse the pre-trained models, we use the earth mover's distance to measure the similarity between the target dataset $\mathcal{D}_T$ and all the synthetic datasets.

(3) Extensive experiments on synthetic and real datasets show that model reuse and fine-tuning can accelerate the building of learned indices by 30.4% and improve lookup efficiency by up to 24.4% on real datasets and 22.5% on skewed synthetic datasets.

II. Related Work

A learned index [1], [3], [4], [10]–[18] learns a mapping from a search key to the storage address of a data record with a machine learning model. Due to limits on the flexibility of a single model, existing learned indices such as RMI [1] build a hierarchy of models to index large datasets. The idea is similar to that of traditional hierarchical indices: top-level models predict partitions of the data records (i.e., the lower-level model in which a record is indexed), while leaf-level models predict the storage locations. The training of a hierarchical learned index can be very expensive, especially when neural networks are used. Follow-up studies aim to bound the prediction error of the learned model. For example, PGM [3] builds a hierarchical learned index bottom up, with a worst-case error bound ϵ on every learned model [19]. RS [4] addresses the training cost problem by training with just a single pass, but it exhibits subpar key-lookup performance compared to state-of-the-art learned indices such as RMI. PLEX [20] improves the robustness of RS and simplifies the parametrization of the radix layer.

A technique related to learning indices is *quantile estimation*, where quantiles are cut points that partition the data domain into disjoint equal-sized intervals. A search key can be located by first locating the partition to which it belongs using the quantiles (e.g., with a binary search) and then performing a search within the partition. Hist-tree [21] integrates such an idea into a tree structure where each node represents a histogram, which partitions a data range into a fixed number of equal-width bins. A learned index can also be thought of as a learned mapping from a search key $p.key$ to a partition $[\mathcal{M}(p.key) - err_l, \mathcal{M}(p.key) + err_u]$, while it does not need to store or search using the quantiles.

Our proposed technique is motivated by domain adaptation – a technique that uses a pre-trained source model $\mathcal{M}_S$ over source dataset $\mathcal{D}_S$, and it fine-tunes $\mathcal{M}_S$ over target dataset $\mathcal{D}_T$ such that $\mathcal{M}_S$ can be used to perform inference on domain $\mathcal{D}_T$. Typically, $\mathcal{D}_T$ has limited data records, which is hard to train, while $\mathcal{D}_S$ is an abundant dataset. Training $\mathcal{M}_S$ is costly but $\mathcal{M}_S$ can well represent $\mathcal{D}_S$. In addition, domain adaptation is limited by how similar the source and target tasks are [22]. In our work, we train a set of models based on very small datasets which can exaggerate the similarity. We fine-tune the models over large datasets, which leads to better initial performance and faster overall training convergence.

Since the key idea of a learned index is learning the CDF [1], we will be using the CDF to measure the similarity between two tasks. The *Kolmogorov–Smirnov* (KS) test uses a similarity measurement which finds the largest absolute difference between two empirical CDFs evaluated at any record. It takes $O(n)$ time to compute, assuming two datasets each with n sorted records. Another measurement is the EMD [8], also known as the *first Wasserstein distance*. The EMD can also be shown to be equal to the area between two empirical CDFs. The cost of computing EMD on one-dimension data is also $O(n)$. In this paper, we choose EMD because KS measures the maximum absolute difference between the CDFs, which is more sensitive to local deformations than EMD (which calculates the average difference). Furthermore, we approximate the CDF with a histogram and use EMD to measure this similarity to reduce the computational cost.

III. Model Reuse and Fine-Tuning

We first present an overview of our model reuse technique in Section III-A. We then detail its key components, including dataset similarity measurement in Section III-B, synthetic dataset generation in Section III-C, model adaptation in Section III-D, and fine-tuning in Section III-E.

A. Solution Overview

Fig. 2 depicts the overall workflow of our proposed solution. First, we propose a heuristic method to generate

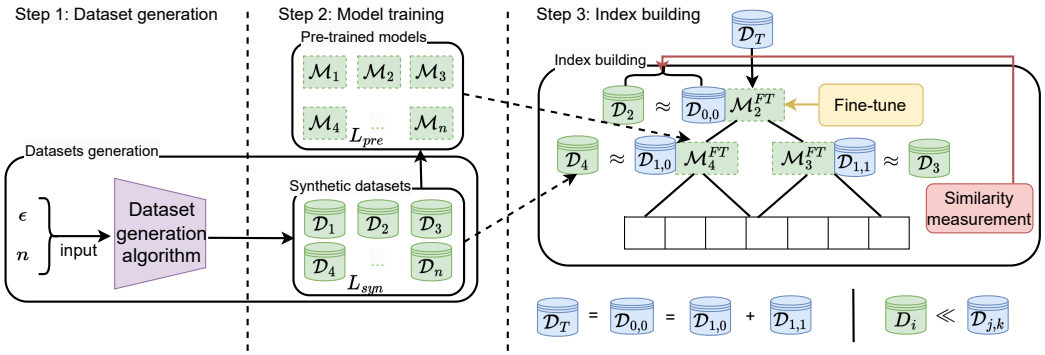


Fig. 2: An overview of dataset generation, model pre-training, and index building. For model reuse and fine-tuning, we use the similarity comparison between $\mathcal{D}_2$ and $\mathcal{D}_{0,0}$ as an example. Then, we fine-tune model $\mathcal{M}_2$ after adaption, which derives $\mathcal{M}_2^{FT}$. Here, $\mathcal{D}_T = \mathcal{D}_{0,0} = \mathcal{D}_{1,0} \cup \mathcal{D}_{1,1}$, i.e., $\mathcal{D}_T$ is $\mathcal{D}_{0,0}$, while $\mathcal{D}_{0,0}$ is separated into $\mathcal{D}_{1,0}$ and $\mathcal{D}_{1,1}$. Any generated synthetic dataset is much smaller than the input datasets in the index building procedure, i.e., $\mathcal{D}_i \ll \mathcal{D}_{j,k}$.

synthetic datasets (Step 1). After acquiring a set of synthetic datasets L_{syn}, we prepare the pre-trained models L_{pre}. Each pre-trained model $L_{pre}[i]$ is trained over a synthetic dataset $L_{syn}[i]$ (Step 2). The preparation process is a one-off work and L_{syn} and L_{pre} can be efficiently loaded for index building.

For index (e.g., RMI [1]) building (Step 3), Algorithm 1 summarizes our procedure. The algorithm scans all the synthetic datasets (line 2) and computes the distance (dissimilarity) between $\mathcal{D}_T$ and each synthetic dataset $L_{syn}[i]$ (line 3), to find the dataset $\mathcal{D}_S$ that has the smallest distance to $\mathcal{D}_T$ (lines 4 and 5). After finding the optimal dataset $\mathcal{D}_S$ and model $\mathcal{M}_S$, we adapt (i.e., to fit the data domain) and fine-tune $\mathcal{M}_S$ based on $\mathcal{D}_S$ and $\mathcal{D}_T$ to obtain $\mathcal{M}_S^{FT}$ (lines 6 and 7). Finally, we compute the error range of $\mathcal{M}_S^{FT}$ on $\mathcal{D}_T$ to bound the index lookup range (line 8) and return $\mathcal{M}_S^{FT}$ afterwards (line 9).

Algorithm 1: Model Reuse and Fine-Tuning

Input: $\mathcal{D}_T$, L_{syn}, L_{pre}
Output: $\mathcal{M}_S^{FT}$

1 $dist_{min} \leftarrow MAX\_INT$;
2 **for** $i \in [1, L_{syn}.size()]$ **do**
3 $\quad dist \leftarrow \texttt{cal\_distance}(L_{syn}[i], \mathcal{D}_T)$;
4 $\quad$ **if** $dist < dist_{min}$ **then**
5 $\quad\quad \mathcal{D}_S \leftarrow L_{syn}[i]$, $\mathcal{M}_S \leftarrow L_{pre}[i]$, $dist_{min} \leftarrow dist$;
6 $\mathcal{M}_S^{FT} \leftarrow \texttt{adapt\_model}(\mathcal{M}_S, \mathcal{D}_S, \mathcal{D}_T)$;
7 $\mathcal{M}_S^{FT} \leftarrow \texttt{fine\_tune}(\mathcal{M}_S^{FT}, \mathcal{D}_T)$;
8 $\mathcal{M}_S^{FT}.errors \leftarrow \mathcal{M}_S^{FT}.\texttt{calc\_err}(\mathcal{D}_T)$;
9 **return** $\mathcal{M}_S^{FT}$;

B. Dataset Similarity Measurement

A model $\mathcal{M}_S$ learns a CDF of dataset $\mathcal{D}_S$. To reuse $\mathcal{M}_S$ on $\mathcal{D}_T$, it is important that the CDFs of $\mathcal{D}_S$ and $\mathcal{D}_T$ are similar. We thus define the distance by the EMD based on the CDFs.

Definition 1 (Similarity between two datasets)**.** *Given two datasets $\mathcal{D}_S$ and $\mathcal{D}_T$, their* similarity *is defined by the area between their empirical CDFs:*

$$\texttt{dist}(\mathcal{D}_S, \mathcal{D}_T) = \int_{-\infty}^{\infty} |cdf_S(x) - cdf_T(x)| dx \qquad (1)$$

However, directly using EMD in our work has two practical issues: 1) The data ranges of $\mathcal{D}_S$ and $\mathcal{D}_T$ can be different, which will impact the accuracy of EMD; 2) The computation cost of EMD is linear to the dataset size n, i.e., $O(n)$, which can affect index build times for larger datasets.

To address these issues, we observe that the similarity measurement is used to choose a pre-trained model, without stringent requirements for high accuracy. Thus, we propose a fast approximation of the similarity metric using *relative frequency histograms* ("histograms" for short), which discretize the data domain into m bins and record relative data frequencies (i.e., percentages) of each bin. A histogram is a discrete approximation of the *probability density function* (PDF) of a dataset. We use it to compute approximations of the CDFs and their distance, denoted by $\texttt{dist}(\mathcal{D}_S, \mathcal{D}_T)$.

To generate the histogram, all the data keys are normalized by min-max normalization, such that bin i contains keys in the range $(\frac{i-1}{m}, \frac{i}{m}]$. To compute $\texttt{dist}(\mathcal{D}_S, \mathcal{D}_T)$, we first compute the histograms of $\mathcal{D}_S$ and $\mathcal{D}_T$, denoted by H_S and H_T. We then go through each bin of H_S and H_T, add up the probabilities at the bins (to approximate cumulative probabilities of the CDFs), and record the maximum difference in the accumulated probabilities of $\mathcal{D}_S$ and $\mathcal{D}_T$.

Algorithm 2 summarizes the computation, where the input histograms H_S and H_T each has m (a system parameter) bins. We use $H_S[i]$ and $H_T[i]$ to denote the i-th bins and their relative frequencies. The sum of the probabilities of the first i bins of H_S and H_T are denoted by P_S and P_T, i.e., $P_S = \sum_{j=1}^{i} H_S[j]$ and $P_T = \sum_{j=1}^{i} H_T[j]$.

Algorithm 2: Approximate EMD

Input: H_S, H_T
Output: $dist$

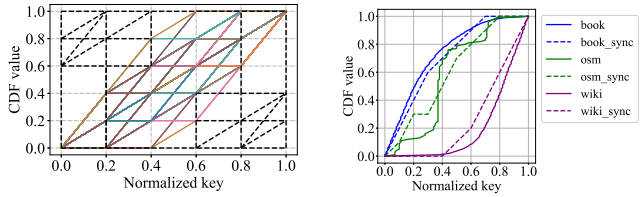
1 $dist \leftarrow 0, P_S \leftarrow 0, P_T \leftarrow 0$;
2 **for** $i \in [1, m]$ **do**
3 $\quad P_S \leftarrow P_S + H_S[i], P_T \leftarrow P_T + H_T[i]$;
4 $\quad dist \leftarrow dist + |P_S - P_T| \cdot \frac{1}{m}$;
5 **return** $dist$;

The algorithm computes $dist$, i.e., an approximation of $\texttt{dist}(\mathcal{D}_S, \mathcal{D}_T)$, by looping through the bins (lines 2 to 4). In the i-th iteration ($i \in [0, m-1]$), it computes $H_S[i] + P_S$. This is the approximate $cdf_S(x)$ for any $x \in (\frac{i-1}{m}, \frac{i}{m}]$ (in our synthetic datasets, $x \in [0, 1]$), because P_S has accumulated the probabilities for $x \leq \frac{i-1}{m}$ while $H_S[i]$ further adds the probability for $x \in (\frac{i-1}{m}, \frac{i}{m}]$. Meanwhile, P_T is the approximate $cdf_T(x)$ for any $x \in (\frac{i-1}{m}, \frac{i}{m}]$. Thus, we use $|P_S - P_T|$ and $|P_S - P_T| \cdot \frac{1}{m}$ to approximate the CDF distance and the difference in the area of bin i, respectively.

Using histograms reduces the similarity computation time to $O(\log |\mathcal{D}_T| + m)$, i.e., $O(\log |\mathcal{D}_T|)$ time for H_T computation and $O(m)$ time for Algorithm 2. Histogram H_S is pre-computed since $\mathcal{D}_S$ is known. Its cost is omitted here.

C. Synthetic Dataset Generation

We aim to generate a set of datasets that can represent any given real dataset with a high similarity. Since it is difficult to determine the position for every single data record in a dataset with large cardinality n, we again approximate the probability density of a dataset by a histogram such that the CDF can be seen as an accumulation of the bins. Then, we can control CDF generation by controlling the number of bins.



(a) The enumeration of all possible CDFs of synthetic datasets ($\epsilon = 0.2$)

(b) CDFs of real and synthetic datasets.

Fig. 3: Examples of the CDFs of synthetic dataset generation and how to approximate real datasets. In (b), the solid lines (e.g., book) are CDFs of real datasets and the dashed lines (e.g., book_sync) are those of synthetic datasets.

To limit the bin value combinations and hence the number of CDFs (synthetic datasets) generated, we use a boundary value ϵ which limits the maximal bin size and limit the probability value of each bin to be within $\{0, \epsilon/2, \epsilon\}$. We use $m = \lceil 2/\epsilon \rceil$ bins in the histogram heuristically. When a target dataset $\mathcal{D}_T$ is matched by a synthetic dataset, their CDF similarity may be within $\epsilon/2$ rather than ϵ, which improves the query performance. Our total number of histograms generated is: $\sum_{i=0}^{m}(C_m^i \cdot C_{m-i}^{\lfloor (1-i\epsilon)/(\epsilon/2) \rfloor})$, where the two combinatorial terms represent the numbers of bins with probability values ϵ and $\epsilon/2$, respectively. Once a histogram is created, we generate a synthetic dataset of n key values ($n = 100$ in our experiments) based on the histogram, where the data range is $[0, 1]$, and random key values are generated for each bin.

As shown in Fig. 3a, after the generation of L_{syn}, all CDFs lie in a $[0, 1] \times [0, 1]$ space. Any CDF can be seen as a curve that starts at $(0, 0)$ and travels to $(1, 1)$ in a non-decreasing manner (in the CDF value dimension). We discretize this space with a grid, where each row has a height of ϵ ($\epsilon = 0.2$ in the figure), and each column has a width of $\lceil 1/\epsilon \rceil$. Consider the set $\mathcal{L}$ of polylines each starting from $(0, 0)$ and traveling to $(1, 1)$ via the grid vertices in a non-decreasing manner (in the CDF value dimension, e.g., the colored lines). In Fig. 3b for all the three CDFs of real datasets, there is a polyline $l \in \mathcal{L}$ such that the distance between l and the CDF is bounded by $\epsilon = 0.2$. However, our synthetic datasets will not cover extremely skewed CDFs (e.g., the black polylines in Fig. 3a).

This procedure is shown to be effective and efficient empirically. We summarize the generation procedure in Algorithm 3. In Fig. 4, we present 12 generated CDFs when $\epsilon = 0.5$.

Algorithm 3: Synthetic Dataset Generation

Input: ϵ, n
Output: L_{syn}

1 $m \leftarrow \lceil 2/\epsilon \rceil$;
2 $bin_h \leftarrow \{0, \epsilon/2, \epsilon\}$;
3 $Set_H \leftarrow$ histograms with m bins and heights in bin_h;
4 **for** $H \in Set_H$ **do**
5 $\mathcal{D} \leftarrow \{\}$;
6 **for** $i \in [0, m-1]$ **do**
7 Randomly generate $H[i] \cdot n$ records in range $(\frac{i}{m}, \frac{i+1}{m}]$ for $\mathcal{D}$;
8 $L_{syn}.add(\mathcal{D})$;
9 **return** L_{syn};

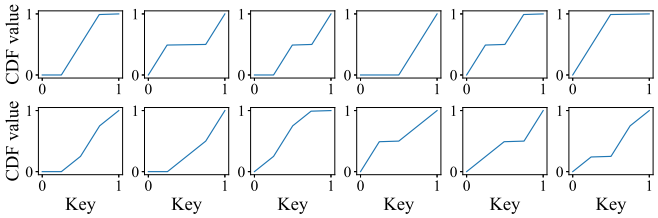


Fig. 4: CDFs of generated synthetic datasets

D. Model adaptation

When model $\mathcal{M}_S$ pre-trained on $\mathcal{D}_S$ is selected, we need to adapt $\mathcal{M}_S$ based on the data domains of $\mathcal{D}_S$ and $\mathcal{D}_T$. This is because $\mathcal{M}_S$ will not work properly on a domain over which it was not trained, even if the CDFs of $\mathcal{D}_S$ and $\mathcal{D}_T$ share a similar shape. Let the data ranges of $\mathcal{D}_S$ and $\mathcal{D}_T$ be $[x_S^s, x_S^e]$ and $[x_T^s, x_T^e]$, and their data storage position ranges be $[y_S^s, y_S^e]$ and $[y_T^s, y_T^e]$, respectively. Model $\mathcal{M}_S$ is trained to take a search key in $[x_S^s, x_S^e]$ as the input and predict a storage position in $[y_S^s, y_S^e]$. Here, we assume that $\mathcal{M}_S$ predicts the storage position of record p directly rather than its rank (or percentile), i.e., $p.addr \approx \mathcal{M}_S(p.key)$ (instead of $\mathcal{M}_S(p.key) \cdot |\mathcal{D}_S|$ as shown in Section I). This simplifies the discussion but does not impact our key findings. To adapt $\mathcal{M}_S$ for $\mathcal{D}_T$, we take a search key in $[x_T^s, x_T^e]$, map it into $[x_S^s, x_S^e]$, and feed the mapped value into $\mathcal{M}_S$ for prediction. The predicted output is mapped back into $[y_T^s, y_T^e]$ for $\mathcal{D}_T$.

Let $S_{\Delta x} = \frac{x_S^e - x_S^s}{x_T^e - x_T^s}$ and $S_{\Delta y} = \frac{y_T^e - y_T^s}{y_S^e - y_S^s}$. The input mapping is performed by a linear transformation $\mathcal{T}_{in}(x) = a_1 \cdot x + b_1$ where $a_1 = S_{\Delta x}$ and $b_1 = x_S^s - x_T^s \cdot S_{\Delta x}$. This is an affine transformation that maps the data range (i.e., $\mathcal{T}_{in}(x_T^s) = x_S^s$ and $\mathcal{T}_{in}(x_T^e) = x_S^e$) without changing the distribution. Similarly, the output mapping is done by $\mathcal{T}_{out}(y) = a_2 \cdot y + b_2$ where $a_2 = S_{\Delta y}$ and $b_2 = y_T^s - y_S^s \cdot S_{\Delta y}$.

E. Fine-Tuning

After model reuse and model adaptation, $\mathcal{M}_S$ is able to index $\mathcal{D}_T$. However, without fine-tuning, the loss may still be high. This is because, while the data distributions are similar in shape, they are not identical. For fine-tuning, we consider both neural networks and linear models. We use θ to represent the parameters of $\mathcal{M}_S$ and $\theta(t)$ denotes the parameters at epoch t during fine-tuning. We use $f(\mathcal{D}_T.key, \theta(t))$ to represent the prediction result of $\mathcal{M}_S$ on $\mathcal{D}_T$. Thus, the loss function $L(\theta)$, which is an L_2 loss, can be written as: $L(\theta) = ||f(\mathcal{D}_T.key, \theta) - \mathcal{D}_T.addr||_2$.

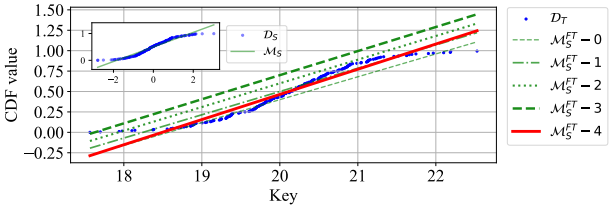


Fig. 5: Model adaptation and fine-tuning, where $\mathcal{M}_S^{FT} - i$ is the ith epoch of fine-tuning ($\mathcal{M}_S^{FT} - 0$ equals to $\mathcal{M}_S$).

Before fine-tuning, $\mathcal{M}_S$ is trained over $\mathcal{D}_S$ such that we denote parameters of $\mathcal{M}_S$ as $\theta(0) = \theta_S$. When we fine-tune $\mathcal{M}_S$, we use gradient descent (GD) with learning rate η and we update the parameters as: $\theta(t+1) = \theta(t) - \eta \frac{\partial L(\theta(t))}{\partial \theta(t)}$, where t denotes the fine-tuning iteration number. In Fig. 5, the inner figure shows a model $\mathcal{M}_S$ selected for $\mathcal{D}_T$. We first adapt the model $\mathcal{M}_S$ from the key range of $\mathcal{D}_S$ to $\mathcal{D}_T$ as $\mathcal{M}_S^{FT}$. Then, we fine-tune $\mathcal{M}_S^{FT}$ with 4 epochs to better fit $\mathcal{D}_T$.

IV. Experiments

A. Experimental Setup

The original implementation of RMI [1] is based on neural networks. However, linear models have lower build costs, which can accelerate both the index build time and lookup time. Thus, we use linear models in the experiments, which are implemented using *Scikit-learn*. All experiments are performed on a 64-bit machine with a 3.60 GHz Intel i9 CPU, RTX 2080Ti GPU, 64 GB RAM, and a 1 TB hard disk drive.

1) Models tested: We compare our approach against both traditional and learned indices: (1) BTree [23] – a C++ based in-memory B$^+$-tree, (2) RMI [1] – the recursive model index using linear models, (3) PGM [3] – a piecewise geometric model index, (4) RS [4] – a single-pass learned index, (5) ALEX [10] – an updatable adaptive learned index, and (6) LIPP [24] – an updatable learned index that yields precise positions of the search keys.

Proposed models. We evaluate the following adapted models equipped with our index building techniques: (1) RMI-MR[2] – RMI enhanced with model reuse (but no fine-tuning), and (2) RMI-MR-FT[2] – RMI-MR with model fine-tuning.

2) Implementation details: The experimental setup follows that of the SOSD benchmark [9].

We summarize the number of synthetic datasets (each with $n = 100$) and the time to pre-train models on them in Table I. When ϵ is smaller than 0.2, the number of bins (i.e., $\lceil 2/\epsilon \rceil$) is getting large such that the number of synthetic datasets explodes. Thus, $\epsilon = 0.2$ is the minimum value that we can use. To balance the computation cost and the fitness of model reuse, we use $\epsilon = 0.3$ as the default value in the later experiments. The pre-trained models and synthetic datasets can be loaded in memory within a second (less than 1 MB in size for linear models); and the total model comparison time to build an index in any of the experiments is also within a second.

For all the baselines, we use their published source code and default configurations. Following the SOSD benchmark, the BTree is built based on sampling data records from an input dataset, where the sampling rate varies from 1 to 2^{-16}. For RMI, we re-implement it using C++ and extend the implementation to integrate model reuse. For fine-tuning, the learning rate is 0.01 and we sample data records with a sampling rate of 0.02. For EMD calculation, we set the number of bins to be $m = 10$ for each histogram.

TABLE I: Summary of Synthetic Datasets

ϵ	0.2	0.3	0.4	0.5
Number of bins (m)	10	7	5	4
Number of datasets	8,953	987	95	19
Model training time (s)	839.5	63.5	8.8	2.1

3) Datasets: Following SOSD, we use four real datasets: `amzn` (default) – an Amazon book popularity dataset, `face` – a Facebook user ID dataset, `osm` – an OpenStreetMap cell ID dataset, and `wiki` – a Wikipedia edit timestamp dataset. We further generate `skewed` datasets from uniform data by raising a key value x to its powers x^α ($\alpha = 3, 5, 7, 9$), following previous works [25], [26]. Each dataset contains 200 million unsigned 64-bit integer keys (1.6 GB in size).

4) Performance metrics: For both real and synthetic datasets, we follow the pareto analysis in SOSD [2], which has ten configurations (i.e., different hyperparameter settings) ranging from minimum to maximum size for each index. For RMI, PGM, and RS, the hyperparameter is the number of models, the threshold of the error bound, and the number of radix bits, respectively. For BTree and ALEX, the hyperparameter is the number of sampled data records used to build the index. LIPP can only be built on the full datasets and hence cannot support the pareto analysis using sampled subsets. Thus, only one data point is recorded for it in each result figure. The SOSD benchmark queries over 10 million sampled keys to show the average lookup (i.e., a query key is in the indexed data) time as the index build time and index size change.

[2] https://anonymous.4open.science/r/MR_FT_on_SOSD-7C1B

B. Results

1) Index size and lookup time over real datasets: Fig. 6 shows that the query efficiency of all learning-based indices is improved as the index size increases. This is because querying over learned indices uses model predictions and then a binary search. A larger number of models can partition the data range into smaller segments, which leads to a smaller search range and decreases the cost of binary search. For BTree, a larger index size means more data records are sampled to build the index, which leads to a trade-off between the tree height and the search range of a leaf node in the tree. Thus, as the index size increases, we first see a drop in lookup time, and then a rise when the index size is larger than about 100 MB.

Under similar index sizes, RMI shows better query performance than ALEX, PGM, and RS in most cases except for `osm` where RS overlaps with RMI in some cases, which is consistent with observations from the SOSD benchmark. RMI-MR and RMI-MR-FT further show slightly better lookup performance than RMI. For example, on `amzn` the improvement by model reuse is up to 20.1% (147 ns vs. 184 ns) and 24.4% (139 ns vs. 184 ns) with further fine-tuning when the index size is 256 MB.

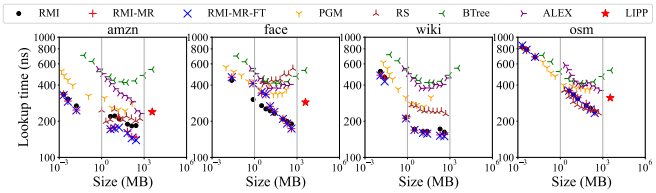


Fig. 6: Index size vs. lookup time over real datasets (LIPP cannot be used on `wiki`)

2) Index build time and lookup time over real datasets: In Fig. 7, we see that RMI-MR dominates RMI in index build times and query efficiency, especially on the `amzn` dataset, where there is a 30.3% (2.29 s vs. 3.29 s) reduction in the index build time. That means, even when RMI trains the linear models with a one pass strategy [9], we can still improve the build time through pre-trained models without jeopardising the query performance. In addition, the query performance can be further improved by fine-tuning (i.e., RMI-MR-FT) because we fine-tune the models with a small number of sampled data records, which brings just a marginal increment in the build time. PGM is slower in lookup under different index size settings. Smaller PGM structures have a large search range, while larger PGM structures have more layers. These lead to high prediction costs. ALEX has an index structure similar to BTree such that they share similar lookup performance.

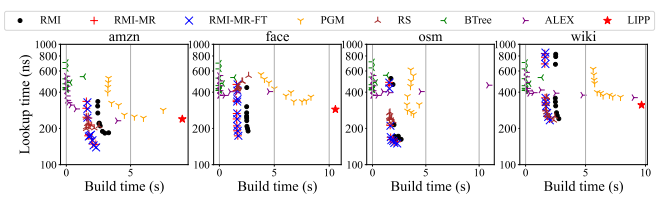


Fig. 7: Build time vs. lookup time over real datasets

3) Index size and lookup time over skewed datasets: Fig. 8 shows that RMI, RMI-MR, and RMI-MR-FT outperform the other methods in lookup time when the index size is large. This is because RMI has just two levels in practice, and the models in the last level (i.e., the leaf level) fit the skewed data records well given a large number of models. Like before, RMI-MR and RMI-MR-FT outperform RMI. For example, when the skewness parameter $\alpha = 3$, the gain in lookup time by model reuse is up to 20.6% (127 ns vs. 160 ns) using an index size of 256 MB. RMI-MR-FT further improves RMI by 22.5% (124 ns vs. 160 ns) in lookup time.

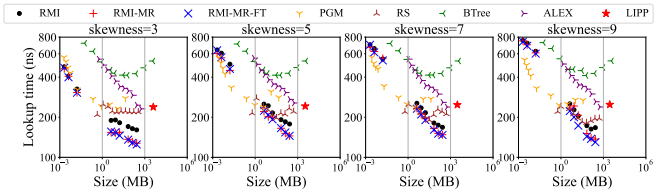


Fig. 8: Index size vs. lookup time over skew datasets

4) Index build time and lookup time over skewed datasets: From Fig. 9, we see that RMI-MR significantly outperforms RMI in both lookup and build times for most cases. The results are similar to those on real datasets. Besides, adding fine-tuning (RMI-MR-FT) slightly improves query efficiency compared to RMI-MR, in the cost of just around 0.1 seconds extra in build times when $\alpha \leq 7$. When $\alpha = 9$, the fine-tuning cost increases given larger index sizes, because there are more models to tune to fit a highly skewed data distribution.

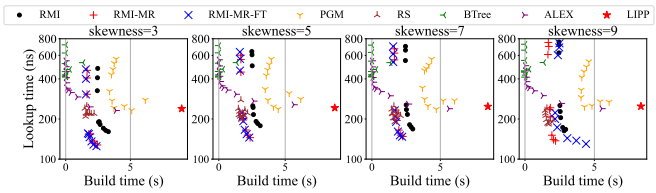


Fig. 9: Build time vs. lookup time over skew datasets

V. Conclusions

We proposed to use pre-trained models and fine-tuning for indexing new datasets to address the building overhead of learned indices. We proposed a CDF based synthetic dataset generation method to generate a number of pre-trained models and use EMD as a similarity metric to select candidate pre-trained models for our learned index. We demonstrated the effectiveness of the proposed techniques by applying them on the RMI learned indices [1]. Experimental results on both synthetic and real data show that our model reuse and fine-tuning techniques can improve the building and the lookup performance of RMI, reducing the index build time by 30.4% on real datasets and 22.5% on skewed datasets.

We plan to extend our techniques to multidimensional data, where the build time for learn indices is typically higher [27].

Acknowledgment

This work is partially supported by Australian Research Council (ARC) Discovery Project DP230101534 as well as Discovery Early Career Researcher Award DE230100366.

REFERENCES

[1] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *SIGMOD*, 2018, pp. 489–504.

[2] R. Marcus, A. Kipf, A. van Renen, M. Stoian, S. Misra, A. Kemper, T. Neumann, and T. Kraska, "Benchmarking learned indexes," *PVLDB*, vol. 14, no. 1, pp. 1–13, 2021.

[3] P. Ferragina and G. Vinciguerra, "The PGM-Index: A fully-dynamic compressed learned index with provable worst-case bounds," *PVLDB*, vol. 13, no. 8, pp. 1162–1175, 2020.

[4] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann, "RadixSpline: A single-pass learned index," in *aiDM*, 2020, pp. 5:1–5.

[5] S. Ben-David, J. Blitzer, K. Crammer, A. Kulesza, F. Pereira, and J. W. Vaughan, "A theory of learning from different domains," *Machine Learning*, vol. 79, no. 1-2, pp. 151–175, 2010.

[6] C. Cortes and M. Mohri, "Domain adaptation in regression," in *International Conference on Algorithmic Learning Theory (ALT)*, 2011, pp. 308–323.

[7] Y. Mansour, M. Mohri, and A. Rostamizadeh, "Domain adaptation: Learning bounds and algorithms," in *COLT*, 2009.

[8] Y. Rubner, C. Tomasi, and L. Guibas, "A metric for distributions with applications to image databases," in *ICCV*, 1998, pp. 59–66.

[9] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann, "SOSD: A benchmark for learned indexes," *NeurIPS Workshop on Machine Learning for Systems*, 2019.

[10] J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann, D. Lomet, and T. Kraska, "ALEX: An updatable adaptive learned index," in *SIGMOD*, 2020, pp. 969–984.

[11] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska, "Fiting-tree: A data-aware index structure," in *SIGMOD*, 2019, pp. 1189–1206.

[12] J. Qi, G. Liu, C. S. Jensen, and L. Kulik, "Effectively learning spatial indices," *PVLDB*, vol. 13, no. 11, pp. 2341–2354, 2020.

[13] A. Davitkova, E. Milchevski, and S. Michel, "The ML-Index: A multidimensional, learned index for point, range, and nearest-neighbor queries," in *EDBT*, 2020, pp. 407–410.

[14] Z. Yang, B. Chandramouli, C. Wang, J. Gehrke, Y. Li, U. F. Minhas, P.-Å. Larson, D. Kossmann, and R. Acharya, "Qd-tree: Learning data layouts for big data analytics," in *SIGMOD*, 2020, pp. 193–208.

[15] Y. Li, D. Chen, B. Ding, K. Zeng, and J. Zhou, "A pluggable learned index method via sampling and gap insertion," *CoRR*, vol. abs/2101.00808, 2021.

[16] V. Pandey, A. van Renen, A. Kipf, I. Sabek, J. Ding, and A. Kemper, "The case for learned spatial indexes," *CoRR*, vol. abs/2008.10349, 2020.

[17] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska, "Learning multi-dimensional indexes," in *SIGMOD*, 2020, pp. 985–1000.

[18] A. Kipf, D. Horn, P. Pfeil, R. Marcus, and T. Kraska, "Lsi: A learned secondary index structure," ser. aiDM, 2022, pp. 4:1–5.

[19] P. Ferragina, F. Lillo, and G. Vinciguerra, "On the performance of learned data structures," *Theoretical Computer Science*, vol. 871, pp. 107–120, 2021.

[20] M. Stoian, A. Kipf, R. Marcus, and T. Kraska, "PLEX: towards practical learned indexing," *CoRR*, vol. abs/2108.05117, 2021.

[21] A. Crotty, "Hist-Tree: Those who ignore it are doomed to learn," in *CIDR*, 2021.

[22] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 10, pp. 1345–1359, 2009.

[23] STX B+ Tree. (2007) https://panthema.net/2007/stx-btree. Accessed: 2022-12-28.

[24] J. Wu, Y. Zhang, S. Chen, J. Wang, Y. Chen, and C. Xing, "Updatable learned index with precise positions," *Proc. VLDB Endow.*, vol. 14, no. 8, p. 1276–1288, 2021.

[25] J. Qi, Y. Tao, Y. Chang, and R. Zhang, "Theoretically optimal and empirically efficient R-trees with strong parallelizability," *PVLDB*, vol. 11, no. 5, pp. 621–634, 2018.

[26] ——, "Packing R-trees with space-filling curves: Theoretical optimality, empirical efficiency, and bulk-loading parallelizability," vol. 45, no. 3, pp. 14:1–14:47, 2020.

[27] G. Liu, J. Qi, C. S. Jensen, J. Bailey, and L. Kulik, "Efficiently learning spatial indices," in *ICDE*, 2023.